4 Programming your infrastructure: The command-line, SDKs, and CloudFormation

This chapter covers

- Understanding the idea of infrastructure as code
- Using the CLI to start a virtual machine
- Using the JavaScript SDK for Node.js to start a virtual machine
- Using CloudFormation to start a virtual machine

Imagine that you want to provide room lighting as a service. To switch off the lights in a room using software, you need a hardware device like a relay connected to the light circuit. This hardware device must have some kind of interface that lets you send commands via software. With a relay and an interface, you can offer room lighting as a service.

This also applies to providing VMs as a service. VMs are software, but if you want to be able to start them remotely, you still need hardware that can handle and fulfill

your request. AWS provides an *application programming interface* (API) that can control every part of AWS over HTTP. Calling the HTTP API is very low-level and requires a lot of repetitive work, like authentication, data (de)serialization, and so on. That's why AWS offers tools on top of the HTTP API that are easier to use. Those tools are:

- *Command-line interface (CLI)*—With one of the CLIs, you can make calls to the AWS API from your terminal.
- *Software development kit (SDK)*—SDKs, available for most programming languages, make it easy to call the AWS API from your programming language of choice.
- *AWS CloudFormation*—Templates are used to describe the state of the infrastructure. AWS CloudFormation translates these templates into API calls.

Not all examples are covered by the Free Tier

The examples in this chapter are *not* all covered by the Free Tier. A special warning message appears when an example incurs costs. As for the other examples, as long as you don't run them longer than a few days, you won't pay anything for them. Keep in mind that this applies only if you created a fresh AWS account for this book and nothing else is going on in your AWS account. Try to complete the chapter within a few days; you'll clean up your account at the end.

On AWS, everything can be controlled via an API. You interact with AWS by making calls to the REST API using the HTTPS protocol, as figure 4.1 illustrates. Everything is available through the API. You can start a virtual machine with a single API call, create 1 TB of storage, or start a Hadoop cluster over the API. By everything, we really mean *everything*. You'll need some time to understand the consequences. By the time you finish this book, you'll ask why the world wasn't always this easy.

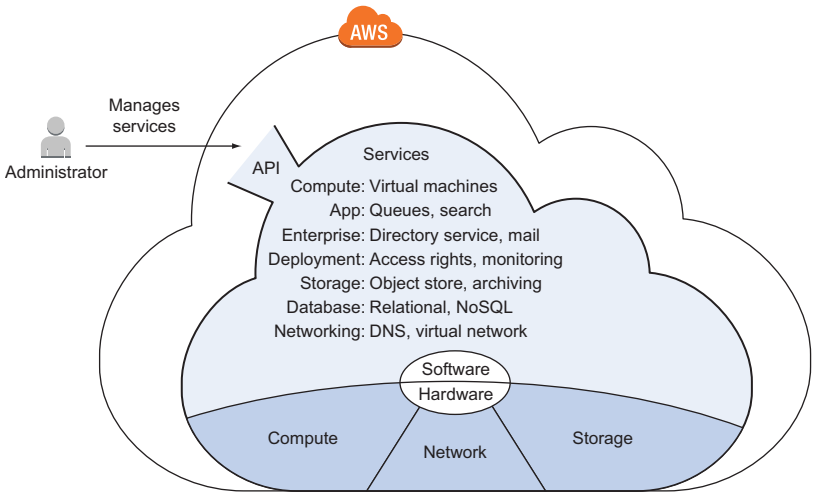
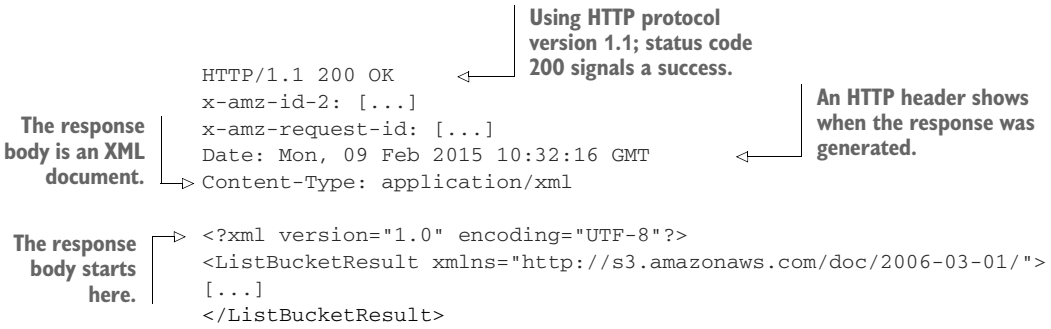


Figure 4.1 Interacting with AWS by making calls to the REST API

Let's look at how the API works. Imagine you uploaded a few files to the object store S3 (you will learn about S3 in chapter 8). Now you want to list all the files in the `S3` object store to check if the upload was successful. Using the raw HTTP API, you send a `GET` request to the API endpoint using the HTTP protocol:

The HTTP response will look like this:



Calling the API directly using plain HTTPS requests is inconvenient. The easy way to talk to AWS is by using the CLI or SDKs, as you learn in this chapter. But the API is the foundation of all those tools.

4.1 Infrastructure as Code

Infrastructure as Code is the idea of using a high-level programming language to control infrastructures. Infrastructure can be any AWS resource, like a network topology, a load balancer, a DNS entry, and so on. In software development, tools like automated tests, code repositories, and build servers increase the quality of software engineering. If your infrastructure is code, you can apply these tools to your infrastructure and improve its quality.

> **WARNING** Don't mix up the terms *Infrastructure as Code* and *Infrastructure as a Service* (IaaS)! IaaS means renting virtual machines, storage, and network with a pay-per-use pricing model.

4.1.1 Automation and the DevOps movement

The *DevOps movement* aims to bring software development and operations together. This usually is accomplished in one of two ways:

- *Using mixed teams with members from both operations and development.* Developers become responsible for operational tasks like being on-call. Operators are involved from the beginning of the software development cycle, which helps make the software easier to operate.
- *Introducing a new role that closes the gap between developers and operators.* This role communicates a lot with both developers and operators, and cares about all topics that touch both worlds.

The goal is to develop and deliver software to the customer rapidly without a negative impact on quality. Communication and collaboration between development and operations are therefore necessary.

The trend toward automation has helped DevOps culture bloom, as it codifies the cooperation between development and operations. You can only do multiple deployments per day if you automate the whole process. If you commit changes to the repository, the source code is automatically built and tested against your automated tests. If the build passes the tests, it's automatically installed in your testing environment. This triggers some acceptance tests. After those tests have been passed, the change is propagated into production. But this isn't the end of the process; now you need to carefully monitor your system and analyze the logs in real time to ensure that the change was successful.

If your infrastructure is automated, you can spawn a new system for every change introduced to the code repository and run the acceptance tests isolated from other changes that were pushed to the repository at the same time. Whenever a change is made to the code, a new system is created (virtual machine, databases, networks, and so on) to run the change in isolation.

4.1.2 Inventing an infrastructure language: JIML

For the purposes of learning infrastructure as code in detail, let's invent a new language to describe infrastructure: JSON Infrastructure Markup Language (JIML). Figure 4.2 shows the infrastructure that will be created in the end.

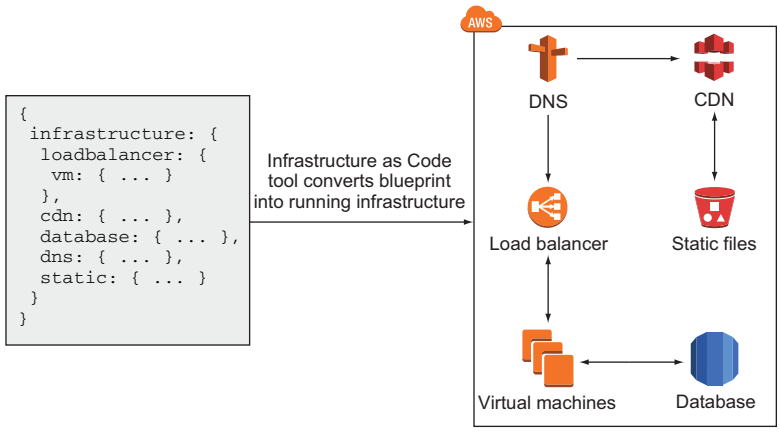


Figure 4.2 From JIML blueprint to infrastructure: infrastructure automation

The infrastructure consists of the following:

- Load balancer (LB)
- Virtual machines (VMs)
- Database (DB)
- DNS entry
- Content delivery network (CDN)
- Bucket for static files

To reduce issues with syntax, let's say JIML is based on JSON. The following JIML code creates the infrastructure shown in figure 4.2. The `$` indicates a reference to an ID.

Listing 4.1 Infrastructure description in JIML

```
{
  "region": "us-east-1",
  "resources": [{
    "type": "loadbalancer",
    "id": "LB",
    "config": {
      "virtualmachines": 2,
      "virtualmachine": {
        "cpu": 2,
        "ram": 4,
        "os": "ubuntu"
      }
    },
    "waitFor": "$DB"
  }, {
    "type": "cdn",
    "id": "CDN",
    "config": {
      "defaultSource": "$LB",
      "sources": [{
        "path": "/static/*",
        "source": "$BUCKET"
      }]
    }
  }, {
    "type": "database",
    "id": "DB",
    "config": {
      "password": "***",
      "engine": "MySQL"
    }
  }, {
    "type": "dns",
    "config": {
      "from": "www.mydomain.com",
      "to": "$CDN"
    }
  }, {
    "type": "bucket",
    "id": "BUCKET"
  }]
}
```

A load balancer is needed.

Need two VMs.

VMs are Ubuntu Linux (4 GB memory, 2 cores).

LB can only be created if the database is ready.

A CDN is used that caches requests to the LB or fetches static assets (images, CSS files, ...) from a bucket.

Data is stored within a MySQL database.

A DNS entry points to the CDN.

A bucket is used to store static assets (images, CSS files, ...).

How can we turn this JSON into AWS API calls?

1. Parse the JSON input.
2. The JIML tool creates a dependency graph by connecting the resources with their dependencies.
3. The JIML tool traverses the dependency graph from the bottom (leaves) to the top (root) and a linear flow of commands. The commands are expressed in a pseudo language.
4. The commands in pseudo language are translated into AWS API calls by the JIML runtime.

The AWS API calls have to be made based on the resources defined in the blueprint. In particular, it is necessary to send the AWS API calls in the correct order. Let's look at the dependency graph created by the JIML tool, shown in figure 4.3.

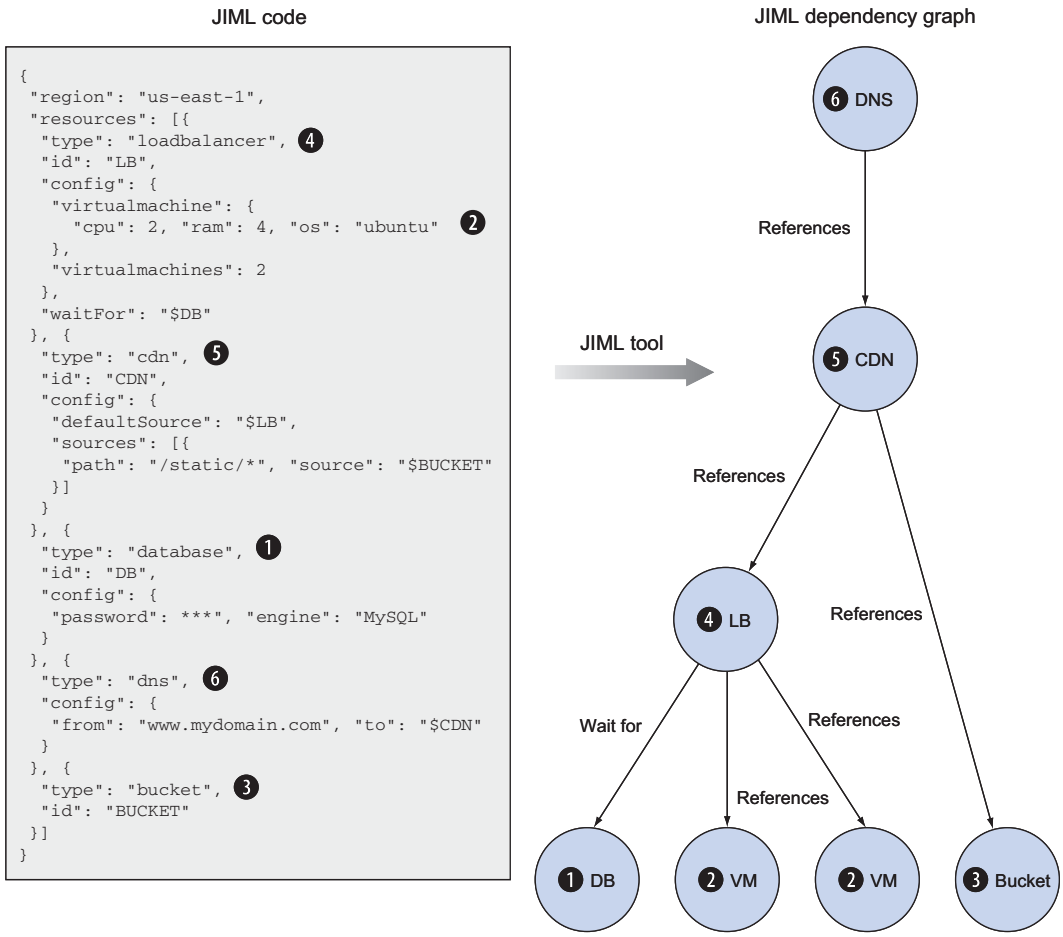


Figure 4.3 The JIML tool figures out the order in which resources need to be created.

You traverse the dependency graph in figure 4.3 from bottom to top and from left to right. The nodes at the bottom have no children: `DB` ❶, `virtualmachine` ❷, and `bucket` ❸. Nodes without children have no dependencies. The `LB` ❹ node depends on the `DB` node and the two `virtualmachine` nodes. The `CDN` ❺ node depends on the `LB` node and the `bucket` ❸ node. Finally, the `DNS` ❻ node depends on the `CDN` node.

The JIML tool turns the dependency graph into a linear flow of commands using pseudo language. The pseudo language represents the steps needed to create all the resources in the correct order. The nodes are easy to create because they have no dependencies, so they're created first.

Listing 4.2 Linear flow of commands in pseudo language, derived from the dependency graph

```
Create the bucket.                                   Create the database.   Create the virtual machine.
$DB = database create {"password": "***", "engine": "MySQL"}
$VM1 = virtualmachine create {"cpu": 2, "ram": 4, "os": "ubuntu"}
$VM2 = virtualmachine create {"cpu": 2, "ram": 4, "os": "ubuntu"}
$BUCKET = bucket create {}

                                  Wait for the dependencies.        Create the load balancer.
await [$DB, $VM1, $VM2]
$LB = loadbalancer create {"virtualmachines": [$VM1, $VM2]}

await [$LB, $BUCKET]
$CDN = cdn create {...}                  <-- Create the CDN.

await $CDN
$DNS = dns create {...}                  <-- Create the DNS entry.

await $DNS
```

We'll skip the last step—translating the commands from the pseudo language into AWS API calls. You've already learned everything you need to know about infrastructure as code: it's all about dependencies.

Now that you know how important dependencies are to infrastructure as code, let's see how you can use the terminal to create infrastructure. The CLI is one tool for implementing infrastructure as code.

4.2 Using the command-line interface

The AWS CLI is a convenient way to use AWS from your terminal. It runs on Linux, macOS, and Windows and is written in Python. It provides a unified interface for all AWS services. Unless otherwise specified, the output is by default in JSON format.

4.2.1 Why should you automate?

Why should you automate instead of using the graphical AWS Management Console? A script or a blueprint can be reused, and will save you time in the long run. You can build new infrastructures quickly with ready-to-use modules from your former projects, or

automate tasks that you will have to do regularly. Automating your infrastructure creation also enhances the automation of your deployment pipeline.

Another benefit is that a script or blueprint is the most accurate documentation you can imagine (even a computer understands it). If you want to reproduce on Monday what you did last Friday, a script is worth its weight in gold. If you're sick and a coworker needs to take care of your tasks, they will appreciate your blueprints.

You're now going to install and configure the CLI. After that, you can get your hands dirty and start scripting.

4.2.2 Installing the CLI

How you proceed depends on your OS. If you're having difficulty installing the CLI, consult http://mng.bz/N8L6 for a detailed description of many installation options.

Linux and macOS

The CLI requires Python (Python 2.6.5+ or Python 3.3+) and pip. pip is the recommended tool for installing Python packages. To check your Python version, run `python --version` in your terminal. If you don't have Python installed or your version is too old, you'll need to install or update Python before continuing with the next step. To find out if you have already installed pip, run `pip --version` in your terminal. If a version appears, you're fine; otherwise, execute the following to install pip:

```
$ curl "https://bootstrap.pypa.io/get-pip.py" -o "get-pip.py"
$ sudo python get-pip.py
```

Verify your pip installation by running `pip --version` in your terminal again. Now it's time to install the AWS CLI:

```
$ sudo pip install awscli
```

Verify your AWS CLI installation by running `aws --version` in your terminal. The version should be at least `1.11.136`.

Windows

The following steps guide you through installing the AWS CLI on Windows using the MSI Installer:

1. Download the AWS CLI (32-bit or 64-bit) MSI installer from http://aws.amazon.com/cli/.
2. Run the downloaded installer, and install the CLI by going through the installation wizard.
3. Run PowerShell as administrator by searching for "PowerShell" in the Start menu and choosing Run as Administrator from its context menu.
4. Type `Set-ExecutionPolicy Unrestricted` into PowerShell, and press Enter to execute the command. This allows you to execute the unsigned PowerShell scripts from our examples.

5 Close the PowerShell window; you no longer need to work as administrator.
6 Run PowerShell by choosing PowerShell from the Start menu.
7 Verify whether the CLI is working by executing `aws --version` in PowerShell. The version should be at least 1.11.136.

4.2.3 Configuring the CLI

To use the CLI, you need to authenticate. Until now, you've been using the root AWS account. This account can do everything, good and bad. It's strongly recommended that you not use the AWS root account (you'll learn more about security in chapter 6), so let's create a new user.

To create a new user, go to at https://console.aws.amazon.com. Click Services in the navigation bar, and click the IAM (AWS Identity and Access Management) service. A page opens as shown in figure 4.4; select Users at left.

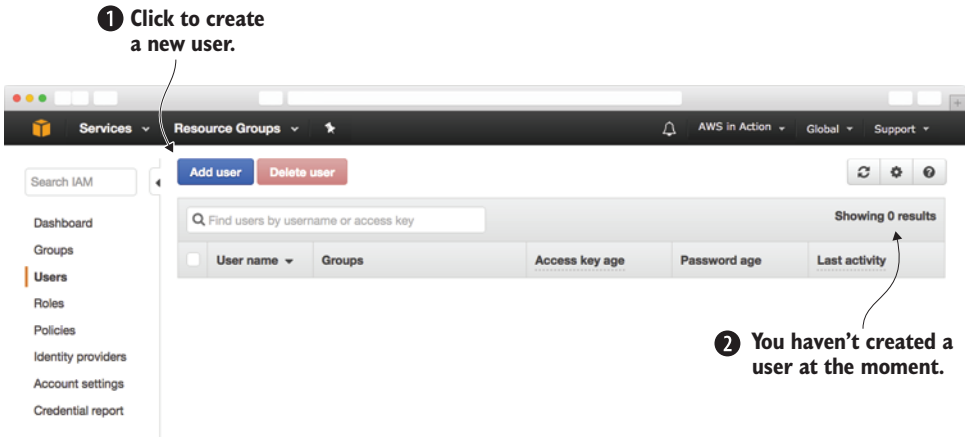


Figure 4.4 IAM users (empty)

Follow these steps to create a user:

1 Click Add User to open the page shown in figure 4.5.
2 Enter `mycli` as the user name.
3 Under Access Type, select Programmatic Access.
4 Click the Next: Permissions button.

In the next step, you have to define the permissions for the new user, as shown in figure 4.6.

1 Click Attach Existing Policies Directly.
2 Select the AdministratorAccess policy.
3 Click the Next: Review button.

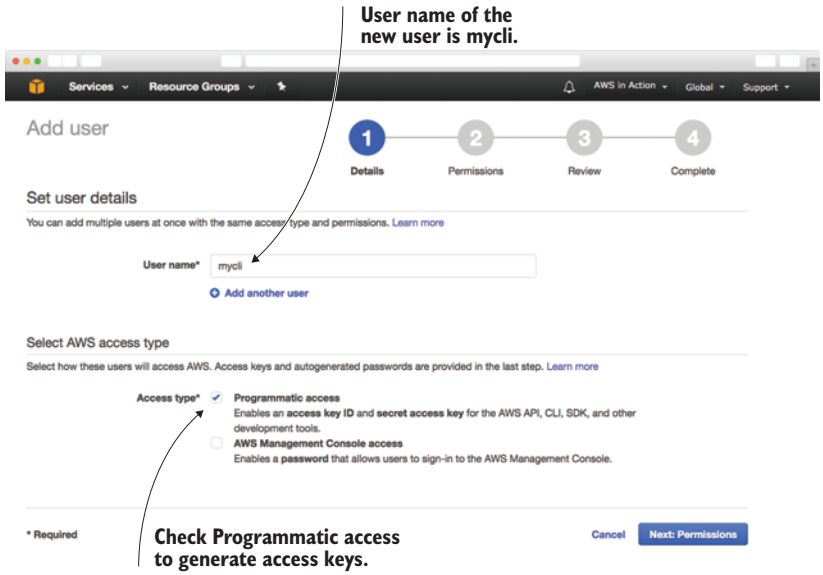


Figure 4.5 Creating an IAM user

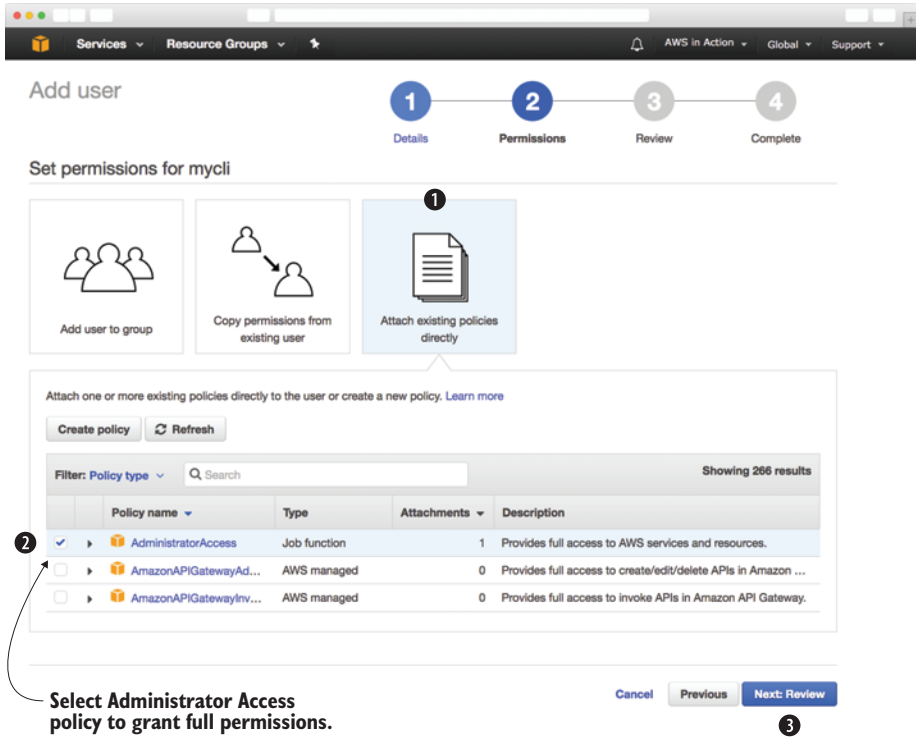


Figure 4.6 Setting permissions for an IAM user

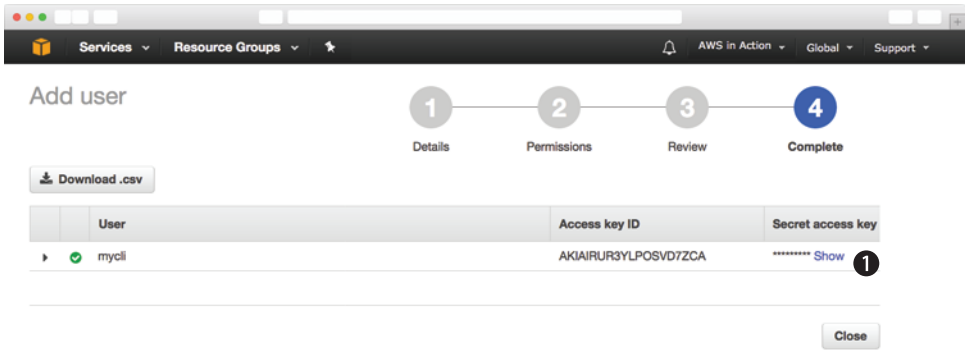


Figure 4.7 Showing access key of an IAM user

The review page sums up what you have configured. Click the Create User button to save. Finally, you will see the page shown in figure 4.7. Click the Show link to display the secret value. You now need to copy the credentials to your CLI configuration. Read on to learn how this works.

Open the terminal on your computer (PowerShell on Windows or a shell on Linux and macOS, not the AWS Management Console), and run `aws configure`. You're asked for four pieces of information:

1. *AWS access key ID*—Copy and paste this value from the Access key ID column (your browser window).
2. *AWS secret access key*—Copy and paste this value from the Secret access key column (your browser window).
3. *Default region name*—Enter `us-east-1`.
4. *Default output format*—Enter `json`.

In the end, the terminal should look similar to this:

Your value will be different! Copy it from your browser window.

```
$ aws configure
AWS Access Key ID [None]:  AKIAIRUR3YLPOSVD7ZCA
AWS Secret Access Key [None]: SSKIng7jkAKERpcT3YphX4cD87sBYgWVw2enqBj7
Default region name [None]: us-east-1
Default output format [None]: json
```

The CLI is now configured to authenticate as the user `mycli`. Switch back to the browser window, and click Close to finish the user-creation wizard.

It's time to test whether the CLI works. Switch to the terminal window, and enter `aws ec2 describe-regions` to get a list of all available regions:

```
$ aws ec2 describe-regions
{
  "Regions": [
    {
      "Endpoint": "ec2.eu-central-1.amazonaws.com",
```

```
      "RegionName": "eu-central-1"
    },
    {
      "Endpoint": "ec2.sa-east-1.amazonaws.com",
      "RegionName": "sa-east-1"
    },
    [...]
    {
      "Endpoint": "ec2.ap-southeast-2.amazonaws.com",
      "RegionName": "ap-southeast-2"
    },
    {
      "Endpoint": "ec2.ap-southeast-1.amazonaws.com",
      "RegionName": "ap-southeast-1"
    }
  ]
}
```

It works! You can now begin to use the CLI.

4.2.4 Using the CLI

Suppose you want to get a list of all running EC2 instances of type t2.micro so you can see what is running in your AWS account. Execute the following command in your terminal:

```
$ aws ec2 describe-instances --filters "Name=instance-type,Values=t2.micro"
{
  "Reservations": []
}
```

Empty list because you haven't created an EC2 instance

To use the AWS CLI, you need to specify a service and an action. In the previous example, the service is `ec2` and the action is `describe-instances`. You can add options with `--key value`:

```
$ aws <service> <action> [--key value ...]
```

One important feature of the CLI is the `help` keyword. You can get help at three levels of detail:

- `aws help`—Shows all available services.
- `aws <service> help`—Shows all actions available for a certain service.
- `aws <service> <action> help`—Shows all options available for the particular service action.

Sometimes you need temporary computing power, like a Linux machine to test something via SSH. To do this, you can write a script that creates a virtual machine for you. The script will run on your local computer and connect to the virtual machine via SSH. After you complete your tests, the script should be able to terminate the virtual machine. The script is used like this:

```
$ ./virtualmachine.sh
waiting for i-c033f117 ...
accepting SSH connections under ec2-54-164-72-62.compute-1.amazonaws.com
ssh -i mykey.pem ec2-user@ec2-54-[...]aws.com
Press [Enter] key to terminate i-c033f117 ...
[...]
terminating i-c033f117 ...
done.
```

Waits until started

SSH connection string

Waits until terminated

Your virtual machine runs until you press the Enter key. When you press Enter, the virtual machine is terminated.

The limitations of this solution are as follows:

- It can handle only one virtual machine at a time.
- There is a different version for Windows than for Linux and macOS.
- It's a command-line application, not graphical.

Nonetheless, the CLI solution solves the following use cases:

- Creating a virtual machine.
- Getting the public name of a virtual machine to connect via SSH.
- Terminating a virtual machine if it's no longer needed.

Depending on your OS, you'll use either Bash (Linux and macOS) or PowerShell (Windows) to script.

One important feature of the CLI needs explanation before you can begin. The `--query` option uses JMESPath, which is a query language for JSON, to extract data from the result. This can be useful because usually you only need a specific field from the result. The following CLI command gets a list of all AMIs in JSON format.

```
$ aws ec2 describe-images
{
  "Images": [
    {
      "ImageId": "ami-146e2a7c",
      "State": "available"
    },
    [...]
    {
      "ImageId": "ami-b66ed3de",
      "State": "available"
    }
  ]
}
```

But to start an EC2 instance, you need the `ImageId` without all the other information. With JMESPath, you can extract just that information. To extract the first `ImageId` property, the path is `Images[0].ImageId`; the result of this query is `"ami-146e2a7c"`.

To extract all `State` properties, the path is `Images[*].State`; the result of this query is `["available", "available"]`.

```
$ aws ec2 describe-images --query "Images[0].ImageId"
"ami-146e2a7c"

$ aws ec2 describe-images --query "Images[0].ImageId" --output text
ami-146e2a7c

$ aws ec2 describe-images --query "Images[*].State"
["available", "available"]
```

With this short introduction to JMESPath, you're well equipped to extract the data you need.

> **Where is the code located?**
> All code can be found in the book's code repository on GitHub: https://github.com/AWSinAction/code2. You can download a snapshot of the repository at https://github.com/AWSinAction/code2/archive/master.zip.

Linux and macOS can interpret Bash scripts, whereas Windows prefers PowerShell scripts. So, we've created two versions of the same script.

Linux and macOS

You can find the following listing in `/chapter04/virtualmachine.sh` in the book's code folder. You can run it either by copying and pasting each line into your terminal or by executing the entire script via `chmod +x virtualmachine.sh && ./virtualmachine.sh`.

Listing 4.3 Creating and terminating a virtual machine from the CLI (Bash)

```
#!/bin/bash -e
AMIID="$(aws ec2 describe-images --filters \
➥ "Name=name,Values=amzn-ami-hvm-2017.09.1.*-x86_64-gp2" \
➥ --query "Images[0].ImageId" --output text)"
VPCID="$(aws ec2 describe-vpcs --filter "Name=isDefault, Values=true" \
➥ --query "Vpcs[0].VpcId" --output text)"
SUBNETID="$(aws ec2 describe-subnets --filters "Name=vpc-id, Values=$VPCID" \
➥ --query "Subnets[0].SubnetId" --output text)"
SGID="$(aws ec2 create-security-group --group-name mysecuritygroup \
➥ --description "My security group" --vpc-id "$VPCID" --output text)"
aws ec2 authorize-security-group-ingress --group-id "$SGID" \
➥ --protocol tcp --port 22 --cidr 0.0.0.0/0
INSTANCEID="$(aws ec2 run-instances --image-id "$AMIID" --key-name mykey \
```

-e makes Bash abort if a command fails.

Get the ID of Amazon Linux AMI.

Get the default VPC ID.

Get the default subnet ID.

Create a Security Group.

Allow inbound SSH connections.

Create and start the virtual machine.

```
    --instance-type t2.micro --security-group-ids "$SGID" \
    --subnet-id "$SUBNETID" --query "Instances[0].InstanceId" --output text)"
echo "waiting for $INSTANCEID ..."
aws ec2 wait instance-running --instance-ids "$INSTANCEID"
PUBLICNAME="$(aws ec2 describe-instances --instance-ids "$INSTANCEID" \
    --query "Reservations[0].Instances[0].PublicDnsName" --output text)"
echo "$INSTANCEID is accepting SSH connections under $PUBLICNAME"
echo "ssh -i mykey.pem ec2-user@$PUBLICNAME"
read -r -p "Press [Enter] key to terminate $INSTANCEID ..."
aws ec2 terminate-instances --instance-ids "$INSTANCEID"
echo "terminating $INSTANCEID ..."
aws ec2 wait instance-terminated --instance-ids "$INSTANCEID"
aws ec2 delete-security-group --group-id "$SGID"
```

Wait until the virtual machine is started.

Get the public name of virtual machine.

Terminate the virtual machine.

Wait until the virtual machine is terminated.

Delete the Security Group.

Cleaning up

Make sure you terminate the virtual machine before you go on!

WINDOWS

You can find the following listing in `/chapter04/virtualmachine.ps1` in the book's code folder. Right-click the virtualmachine.ps1 file, and select Run with PowerShell to execute the script.

Listing 4.4 Creating and terminating a virtual machine from the CLI (PowerShell)

```
$ErrorActionPreference = "Stop"
$AMIID=aws ec2 describe-images --filters \
    "Name=name,Values=amzn-ami-hvm-2017.09.1.*-x86_64-gp2" \
    --query "Images[0].ImageId" --output text
$VPCID=aws ec2 describe-vpcs --filter "Name=isDefault, Values=true" \
    --query "Vpcs[0].VpcId" --output text
$SUBNETID=aws ec2 describe-subnets --filters "Name=vpc-id, Values=$VPCID" \
    --query "Subnets[0].SubnetId" --output text
$SGID=aws ec2 create-security-group --group-name mysecuritygroup \
    --description "My security group" --vpc-id $VPCID \
    --output text
aws ec2 authorize-security-group-ingress --group-id $SGID \
    --protocol tcp --port 22 --cidr 0.0.0.0/0
$INSTANCEID=aws ec2 run-instances --image-id $AMIID --key-name mykey \
    --instance-type t2.micro --security-group-ids $SGID \
    --subnet-id $SUBNETID \
    --query "Instances[0].InstanceId" --output text
```

Get the default subnet ID.

Abort if the command fails.

Get the ID of Amazon Linux AMI.

Get the default VPC ID.

Create the Security Group.

Allow inbound SSH connections.

Create and start the virtual machine.

```
Write-Host "waiting for $INSTANCEID ..."
aws ec2 wait instance-running --instance-ids $INSTANCEID
$PUBLICNAME=aws ec2 describe-instances --instance-ids $INSTANCEID \
➥ --query "Reservations[0].Instances[0].PublicDnsName" --output text
Write-Host "$INSTANCEID is accepting SSH under $PUBLICNAME"
Write-Host "connect to $PUBLICNAME via SSH as user ec2-user"
Write-Host "Press [Enter] key to terminate $INSTANCEID ..."
Read-Host
aws ec2 terminate-instances --instance-ids $INSTANCEID
Write-Host "terminating $INSTANCEID ..."
aws ec2 wait instance-terminated --instance-ids $INSTANCEID
aws ec2 delete-security-group --group-id $SGID
```

Wait until the virtual machine is started.

Get the public name of virtual machine.

Terminate the virtual machine.

Wait until the virtual machine is terminated

Delete the Security Group.

Cleaning up

Make sure you terminate the virtual machine before you go on!

4.3 Programming with the SDK

AWS offers SDKs for a number of programming languages and platforms:

- Android
- Browsers (JavaScript)
- iOS
- Java
- .NET
- Node.js (JavaScript)
- PHP
- Python
- Ruby
- Go
- C++

An AWS SDK is a convenient way to make calls to the AWS API from your favorite programming language. The SDK takes care of things like authentication, retry on error, HTTPS communication, and XML or JSON (de)serialization. You're free to choose the SDK for your favorite language, but in this book most examples are written in JavaScript and run in the Node.js runtime environment.

Installing and getting started with Node.js

Node.js is a platform for executing JavaScript in an event-driven environment so you can easily build network applications. To install Node.js, visit https://nodejs.org and download the package that fits your OS. All examples in this book are tested with Node.js 8.

After Node.js is installed, you can verify if everything works by typing `node --version` into your terminal. Your terminal should response with something similar to `v8.*`. Now you're ready to run JavaScript examples, like the Node Control Center for AWS.

(continued)

Your Node.js installation comes with a important tool called *npm*, which is the package manager for Node.js. Verify the installation by running `npm --version` in your terminal.

To run a JavaScript script in Node.js, enter `node` *script.js* in your terminal, where *script.js* is the name of the script file. We use Node.js in this book because it's easy to install, it requires no IDE, and the syntax is familiar to most programmers.

Don't be confused by the terms JavaScript and Node.js. If you want to be precise, JavaScript is the language and Node.js is the runtime environment. But don't expect anybody to make that distinction. Node.js is also called node.

Do you want to get started with Node.js? We recommend *Node.js in Action (2nd Edition)* from Alex Young, et al. (Manning, 2017), or the video course *Node.js in Motion* from PJ Evans, (Manning, 2018).

To understand how the AWS SDK for Node.js (JavaScript) works, let's create a Node.js (JavaScript) application that controls EC2 instances via the AWS SDK.

4.3.1 Controlling virtual machines with SDK: nodecc

The *Node Control Center for AWS (nodecc)* is for managing multiple temporary EC2 instances using a text UI written in JavaScript. nodecc has the following features:

- It can handle multiple virtual machines.
- It's written in JavaScript and runs in Node.js, so it's portable across platforms.
- It uses a textual UI.

Figure 4.8 shows what nodecc looks like. You can find the nodecc application at `/chapter04/nodecc/` in the book's code folder. Switch to that directory, and run `npm install` in your terminal to install all needed dependencies. To start nodecc, run

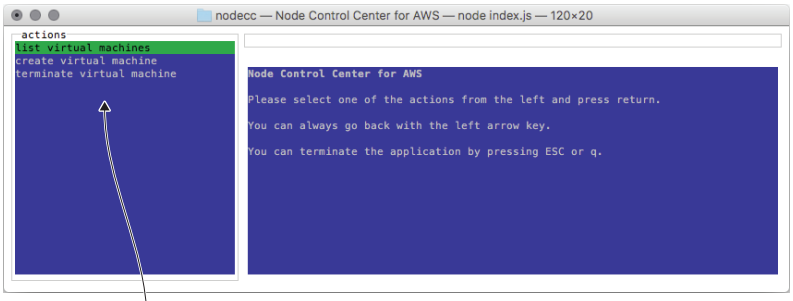


Figure 4.8 nodecc: start screen

`node index.js`. You can always go back by pressing the left arrow key. You can quit the application by pressing Esc or q. The SDK uses the same settings you created for the CLI, so you're using the `mycli` user when running nodecc.

4.3.2 How nodecc creates a virtual machine

Before you can do anything with nodecc, you need at least one virtual machine. To start a virtual machine, choose the AMI you want, as figure 4.9 shows.

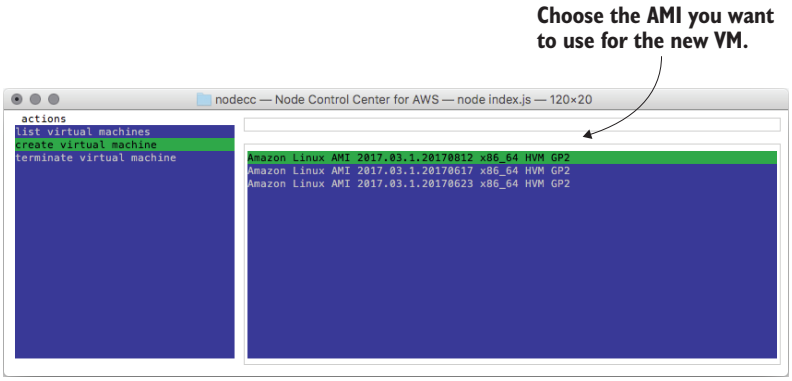


Figure 4.9 nodecc: creating a virtual machine (step 1 of 2)

The code that fetches the list of the available AMIs is located at `lib/listAMIs.js`.

Listing 4.5 Fetching the list of available AMIs /lib/listAMIs.js

```
const jmespath = require('jmespath');
const AWS = require('aws-sdk');
const ec2 = new AWS.EC2({region: 'us-east-1'});

module.exports = (cb) => {
  ec2.describeImages({
    Filters: [{
      Name: 'name',
      Values: ['amzn-ami-hvm-2017.09.1.*-x86_64-gp2']
    }]
  }, (err, data) => {
    if (err) {
      cb(err);
    } else {
      const amiIds = jmespath.search(data, 'Images[*].ImageId');
      const descriptions = jmespath.search(data, 'Images[*].Description');
      cb(null, {amiIds: amiIds, descriptions: descriptions});
    }
  });
};
```

Configure an EC2 endpoint.

Require is used to load modules.

module.exports makes this function available to users of the listAMIs module.

Action

In case of failure, err is set.

Otherwise, data contains all AMIs.

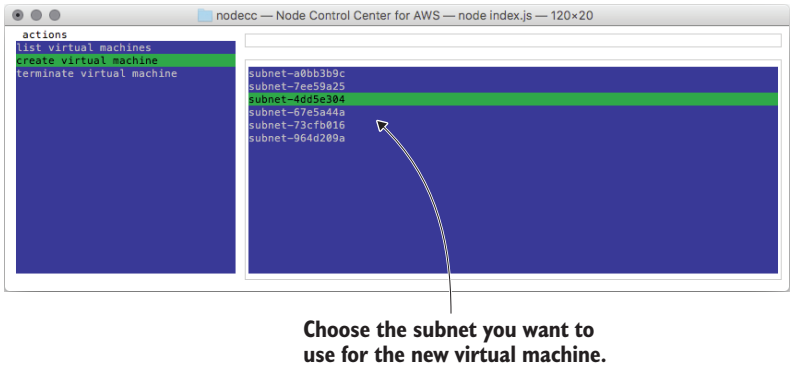


Figure 4.10 nodecc: creating a virtual machine (step 2 of 2)

The code is structured in such a way that each action is implemented in the lib folder. The next step to create a virtual machine is to choose which subnet the virtual machine should be started in. You haven't learned about subnets yet, so for now select one randomly; see figure 4.10. The corresponding script is located at `lib/listSubnets.js`.

After you select the subnet, the virtual machine is created by `lib/createVM.js`, and you see a starting screen. Now it's time to find out the public name of the newly created virtual machine. Use the left arrow key to switch to the navigation section.

4.3.3 How nodecc lists virtual machines and shows virtual machine details

One important use case that nodecc must support is showing the public name of a VM that you can use to connect via SSH. Because nodecc handles multiple virtual machines, the first step is to select a VM, as shown in figure 4.11.

Look at `lib/listVMs.js` to see how a list of virtual machines can be retrieved with the AWS SDK. After you select the VM, you can display its details; see figure 4.12. You could use the `PublicDnsName` to connect to the EC2 instance via SSH. Press the left arrow key to switch back to the navigation section.

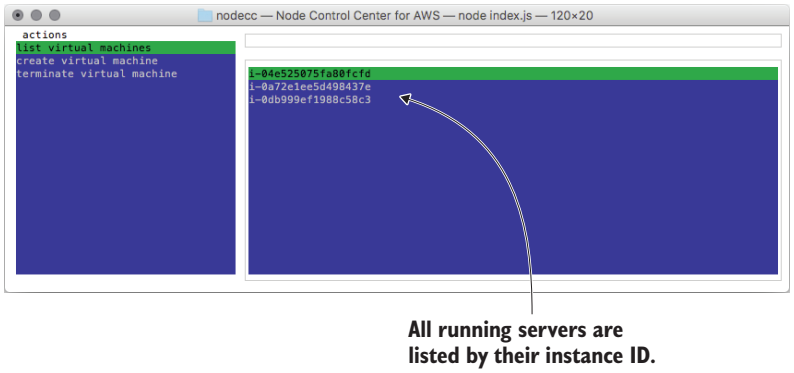


Figure 4.11 nodecc: listing virtual machines

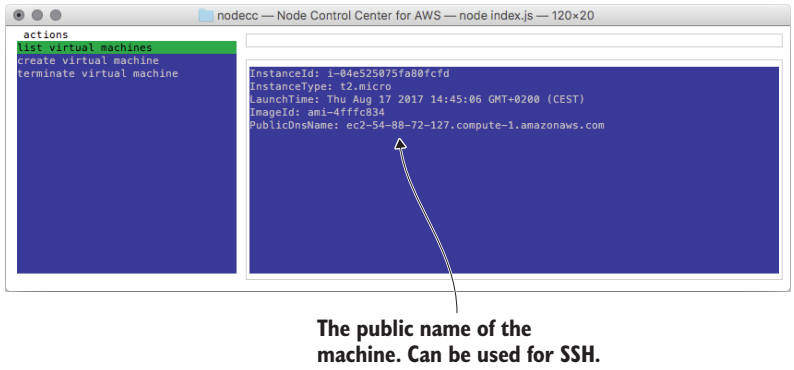


Figure 4.12 nodecc: showing virtual machine details

4.3.4 How nodecc terminates a virtual machine

To terminate a virtual machine, you first have to select it. To list the virtual machines, use `lib/listVMs.js` again. After the VM is selected, `lib/terminateVM.js` takes care of termination.

That's nodecc: a text UI program for controlling temporary EC2 instances. Take some time to think about what you could create by using your favorite language and the AWS SDK. Chances are high that you might come up with a new business idea!

Cleaning up

Make sure you terminate all started virtual machines before you go on!

4.4 Using a blueprint to start a virtual machine

Earlier, we talked about JIML to introduce the concept of infrastructure as code. Luckily, AWS already offers a tool that does much better than JIML: *AWS CloudFormation*. CloudFormation is based on templates, which up to now we've called blueprints.

NOTE We use the term *blueprint* when discussing infrastructure automation in general. Blueprints used for AWS CloudFormation, a configuration management service, are called *templates*.

A *template* is a description of your infrastructure, written in JSON or YAML, that can be interpreted by CloudFormation. The idea of describing something rather than listing the necessary actions is called a *declarative approach*. Declarative means you tell CloudFormation how your infrastructure should look. You aren't telling CloudFormation what actions are needed to create that infrastructure, and you don't specify the sequence in which the actions need to be executed.

The benefits of CloudFormation are as follows:

- *It's a consistent way to describe infrastructure on AWS.* If you use scripts to create your infrastructure, everyone will solve the same problem differently. This is a hurdle for new developers and operators trying to understand what the code is doing. CloudFormation templates are a clear language for defining infrastructure.
- *It can handle dependencies.* Ever tried to register a web server with a load balancer that wasn't yet available? At first glance, you'll miss a lot of dependencies. Trust us: never try to set up complex infrastructure using scripts. You'll end up in dependency hell!
- *It's replicable.* Is your test environment an exact copy of your production environment? Using CloudFormation, you can create two identical infrastructures and keep them in sync.
- *It's customizable.* You can insert custom parameters into CloudFormation to customize your templates as you wish.
- *It's testable.* If you can create your architecture from a template, it's testable. Just start a new infrastructure, run your tests, and shut it down again.
- *It's updatable.* CloudFormation supports updates to your infrastructure. It will figure out the parts of the template that have changed and apply those changes as smoothly as possible to your infrastructure.
- *It minimizes human failure.* CloudFormation doesn't get tired—even at 3 a.m.
- *It's the documentation for your infrastructure.* A CloudFormation template is a JSON or YAML document. You can treat it as code and use a version control system like Git to keep track of the changes.
- *It's free.* Using CloudFormation comes at no additional charge. If you subscribe to an AWS support plan, you also get support for CloudFormation.

We think CloudFormation is one of the most powerful tools available for managing infrastructure on AWS.

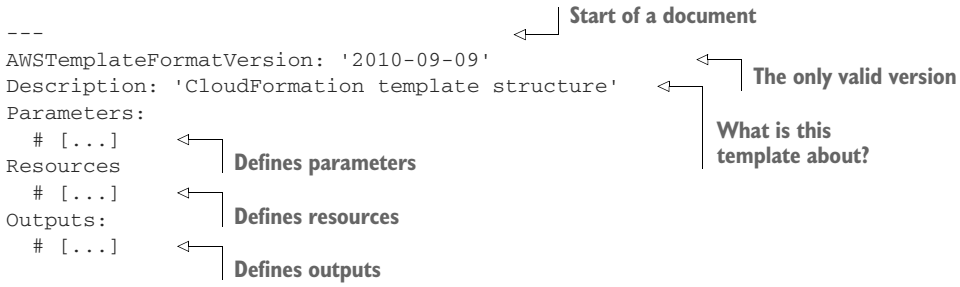
4.4.1 Anatomy of a CloudFormation template

A basic CloudFormation template is structured into five parts:

1. *Format version*—The latest template format version is 2010-09-09, and this is currently the only valid value. Specify this version; the default is to use the latest version, which will cause problems if new versions are introduced in the future.
2. *Description*—What is this template about?
3. *Parameters*—Parameters are used to customize a template with values: for example, domain name, customer ID, and database password.
4. *Resources*—A resource is the smallest block you can describe. Examples are a virtual machine, a load balancer, or an Elastic IP address.
5. *Outputs*—An output is comparable to a parameter, but the other way around. An output returns something from your template, such as the public name of an EC2 instance.

A basic template looks like the following listing.

Listing 4.6 CloudFormation template structure



Let's take a closer look at parameters, resources, and outputs.

Format version and description

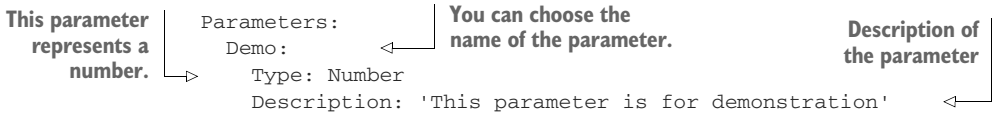
The only valid `AWSTemplateFormatVersion` value at the moment is `2010-09-09`. Always specify the format version. If you don't, CloudFormation will use whatever version is the latest one. As mentioned earlier, this means that if a new format version is introduced in the future, you'll end up using the wrong format and get into serious trouble.

`Description` isn't mandatory, but we encourage you to take some time to document what the template is about. A meaningful description will help you in the future to remember what the template is for. It will also help your coworkers.

Parameters

A parameter has at least a name and a type. We encourage you to add a description as well.

Listing 4.7 CloudFormation parameter structure



Valid types are listed in table 4.1.

Table 4.1 CloudFormation parameter types

Type	Description
`String` `CommaDelimitedList`	A string or a list of strings separated by commas
`Number` `List<Number>`	An integer or float, or a list of integers or floats

Table 4.1 CloudFormation parameter types *(continued)*

Type	Description
`AWS::EC2::AvailabilityZone::Name` `List<AWS::EC2::AvailabilityZone::Name>`	An Availability Zone, such as us-west-2a, or a list of Availability Zones
`AWS::EC2::Image::Id` `List<AWS::EC2::Image::Id>`	An AMI ID or a list of AMIs
`AWS::EC2::Instance::Id` `List<AWS::EC2::Instance::Id>`	An EC2 instance ID or a list of EC2 instance IDs
`AWS::EC2::KeyPair::KeyName`	An Amazon EC2 key-pair name
`AWS::EC2::SecurityGroup::Id` `List<AWS::EC2::SecurityGroup::Id>`	A security group ID or a list of security group IDs
`AWS::EC2::Subnet::Id` `List<AWS::EC2::Subnet::Id>`	A subnet ID or a list of subnet IDs
`AWS::EC2::Volume::Id` `List<AWS::EC2::Volume::Id>`	An EBS volume ID (network attached storage) or a list of EBS volume IDs
`AWS::EC2::VPC::Id` `List<AWS::EC2::VPC::Id>`	A VPC ID (virtual private cloud) or a list of VPC IDs
`AWS::Route53::HostedZone::Id` `List<AWS::Route53::HostedZone::Id>`	A DNS zone ID or a list of DNS zone IDs

In addition to using the `Type` and `Description` properties, you can enhance a parameter with the properties listed in table 4.2.

Table 4.2 CloudFormation parameter properties

Property	Description	Example
`Default`	A default value for the parameter	`Default: 'm5.large'`
`NoEcho`	Hides the parameter value in all graphical tools (useful for passwords)	`NoEcho: true`
`AllowedValues`	Specifies possible values for the parameter	`AllowedValues: [1, 2, 3]`
`AllowedPattern`	More generic than `AllowedValues` because it uses a regular expression	`AllowedPattern: '[a-zA-Z0-9]*'` allows only a–z, A–Z, and 0–9 with any length
`MinLength`, `MaxLength`	Defines how long a parameter can be	`MinLength: 12`
`MinValue`, `MaxValue`	Used in combination with the `Number` type to define lower and upper bounds	`MaxValue: 10`
`ConstraintDescription`	A string that explains the constraint when the constraint is violated.	`ConstraintDescription: 'Maximum value is 10.'`

A parameter section of a CloudFormation template could look like this:

```
Parameters:
  KeyName:
    Description: 'Key Pair name'
    Type: 'AWS::EC2::KeyPair::KeyName'    <-- Only key pair names are allowed.
  NumberOfVirtualMachines:
    Description: 'How many virtual machine do you like?'
    Type: Number
    Default: 1                <-- The default is one virtual machine.
    MinValue: 1
    MaxValue: 5           <-- Prevent massive costs with an upper bound.
  WordPressVersion:
    Description: 'Which version of WordPress do you want?'
    Type: String
    AllowedValues: ['4.1.1', '4.0.1']      <-- Restricted to certain versions
```

Now you should have a better feel for parameters. If you want to know everything about them, see the documentation at http://mng.bz/jg7B or follow along in the book and learn by doing.

Resources

A resource has at least a name, a type, and some properties.

Listing 4.8 CloudFormation resources structure

```
Resources:
  VM:                                  <-- Name or logical ID of the resource that you can choose
    Type: 'AWS::EC2::Instance'         <-- Defines an EC2 instance
    Properties:
      # [...]                          <-- Properties needed for the type of resource
```

When defining resources, you need to know about the type and that type's properties. In this book, you'll get to know a lot of resource types and their respective properties. An example of a single EC2 instance follows. If you see `!Ref` *NameOfSomething*, think of it as a placeholder for what is referenced by the name. You can reference parameters and resources to create dependencies.

Listing 4.9 CloudFormation EC2 instance resource

```
Resources:
  VM:                                  <-- Name or logical ID of the resource that you can choose
    Type: 'AWS::EC2::Instance'         <-- Defines an EC2 instance
    Properties:
      ImageId: 'ami-6057e21a'          <-- Some hard-coded settings
      InstanceType: 't2.micro'
      KeyName: mykey
      NetworkInterfaces:
       - AssociatePublicIpAddress: true
```

```
        DeleteOnTermination: true
        DeviceIndex: 0
        GroupSet:
        - 'sg-123456'
        SubnetId: 'subnet-123456'
```

Now you've described the virtual machine, but how can you output its public name?

OUTPUTS

A CloudFormation template's output includes at least a name (like parameters and resources) and a value, but we encourage you to add a description as well. You can use outputs to pass data from within your template to the outside.

Listing 4.10 CloudFormation outputs structure

```
Outputs:
  NameOfOutput:
    Value: '1'
    Description: 'This output is always 1'
```

Value of the output

Name of the output that you can choose

Static outputs like this one aren't very useful. You'll mostly use values that reference the name of a resource or an attribute of a resource, like its public name.

Listing 4.11 CloudFormation outputs example

```
Outputs:
  ID:
    Value: !Ref Server
    Description: 'ID of the EC2 instance'
  PublicName:
    Value: !GetAtt 'Server.PublicDnsName'
    Description: 'Public name of the EC2 instance'
```

References the EC2 instance

Get the attribute PublicDnsName of the EC2 instance.

You'll get to know the most important attributes of `!GetAtt` later in the book. If you want to know about all of them, see http://mng.bz/q5I4.

Now that we've taken a brief look at the core parts of a CloudFormation template, it's time to make one of your own.

4.4.2 Creating your first template

How do you create a CloudFormation template? Different options are available:

- Use a text editor or IDE to write a template from scratch.
- Use the CloudFormation Designer, a graphical user interface offered by AWS.
- Start with a template from a public library that offers a default implementation and adapt it to your needs.
- Use CloudFormer, a tool for creating a template based on an existing infrastructure provided by AWS.
- Use a template provided by your vendor.

AWS and their partners offer CloudFormation templates for deploying popular solutions: AWS Quick Starts at https://aws.amazon.com/quickstart/. Furthermore, we have open sourced the templates we are using in our day-to-day work on GitHub: https://github.com/widdix/aws-cf-templates.

Suppose you've been asked to provide a VM for a developer team. After a few months, the team realizes the VM needs more CPU power, because the usage pattern has changed. You can handle that request with the CLI and the SDK, but as you learned in section 3.4, before the instance type can be changed, you must stop the EC2 instance. The process will be as follows:

1. Stop the instance.
2. Wait for the instance to stop.
3. Change the instance type.
4. Start the instance.
5. Wait for the instance to start.

A declarative approach like that used by CloudFormation is simpler: just change the `InstanceType` property and update the template. `InstanceType` can be passed to the template via a parameter. That's it! You can begin creating the template.

Listing 4.12 Template to create an EC2 instance with CloudFormation

```
---
AWSTemplateFormatVersion: '2010-09-09'
Description: 'AWS in Action: chapter 4'
Parameters:
  KeyName:
    Description: 'Key Pair name'
    Type: 'AWS::EC2::KeyPair::KeyName'
    Default: mykey
  VPC:
    # [...]
  Subnet:
    # [...]
  InstanceType:
    Description: 'Select one of the possible instance types'
    Type: String
    Default: 't2.micro'
    AllowedValues": ['t2.micro', 't2.small', 't2.medium']
Resources:
  SecurityGroup:
    Type: 'AWS::EC2::SecurityGroup'
    Properties:
      # [...]
  VM:
    Type: 'AWS::EC2::Instance'
    Properties:
      ImageId: 'ami-6057e21a'
      InstanceType: !Ref InstanceType
      KeyName: !Ref KeyName
      NetworkInterfaces:
```

The user defines which key to use.

You'll learn about this in section 6.5.

You'll learn about this in section 6.5.

The user defines the instance type.

You'll learn about this in section 6.4.

Defines a minimal EC2 instance

```
        - AssociatePublicIpAddress: true
          DeleteOnTermination: true
          DeviceIndex: 0
          GroupSet:
          - !Ref SecurityGroup
          SubnetId: !Ref Subnet
Outputs:                                    Returns the public name
  PublicName:                               of the EC2 instance
    Value: !GetAtt 'Server.PublicDnsName'
    Description: 'Public name (connect via SSH as user ec2-user)'
```

You can find the full code for the template at `/chapter04/virtualmachine.yaml` in the book's code folder. Please don't worry about VPC, subnets, and security groups at the moment; you'll get to know them in chapter 6.

Where is the template located?

You can find the template on GitHub. You can download a snapshot of the repository at https://github.com/AWSinAction/code2/archive/master.zip. The file we're talking about is located at `chapter04/virtualmachine.yaml`. On S3, the same file is located at http://mng.bz/B5UM.

If you create an infrastructure from a template, CloudFormation calls it a *stack*. You can think of *template* versus *stack* much like *class* versus *object*. The template exists only once, whereas many stacks can be created from the same template.

Open the AWS Management Console at https://console.aws.amazon.com. Click Services in the navigation bar, and then click the CloudFormation service. Figure 4.13 shows the initial CloudFormation screen with an overview of all the stacks.

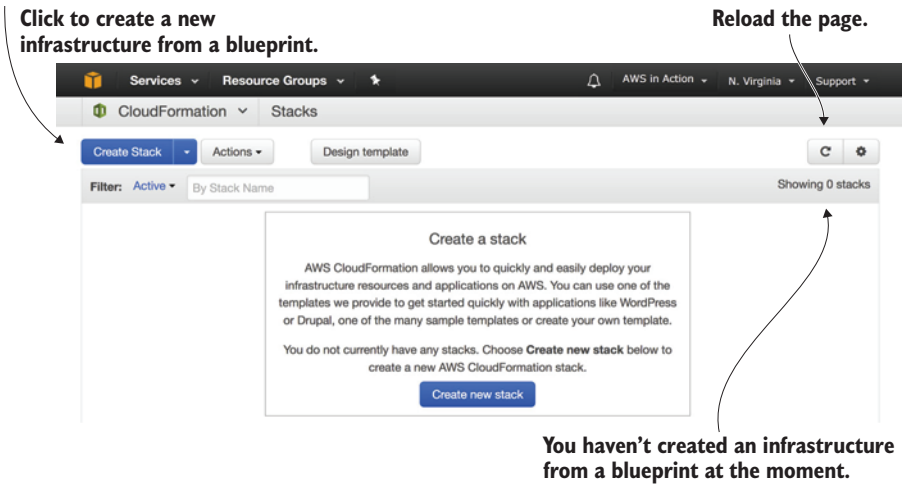


Figure 4.13 Overview of CloudFormation stacks

The following steps will guide you through creating your stack:

1. Click Create Stack to start a four-step wizard.
2. Select Specify an Amazon S3 Template URL, and enter `https://s3.amazonaws.com/awsinaction-code2/chapter04/virtualmachine.yaml` as shown in figure 4.14.

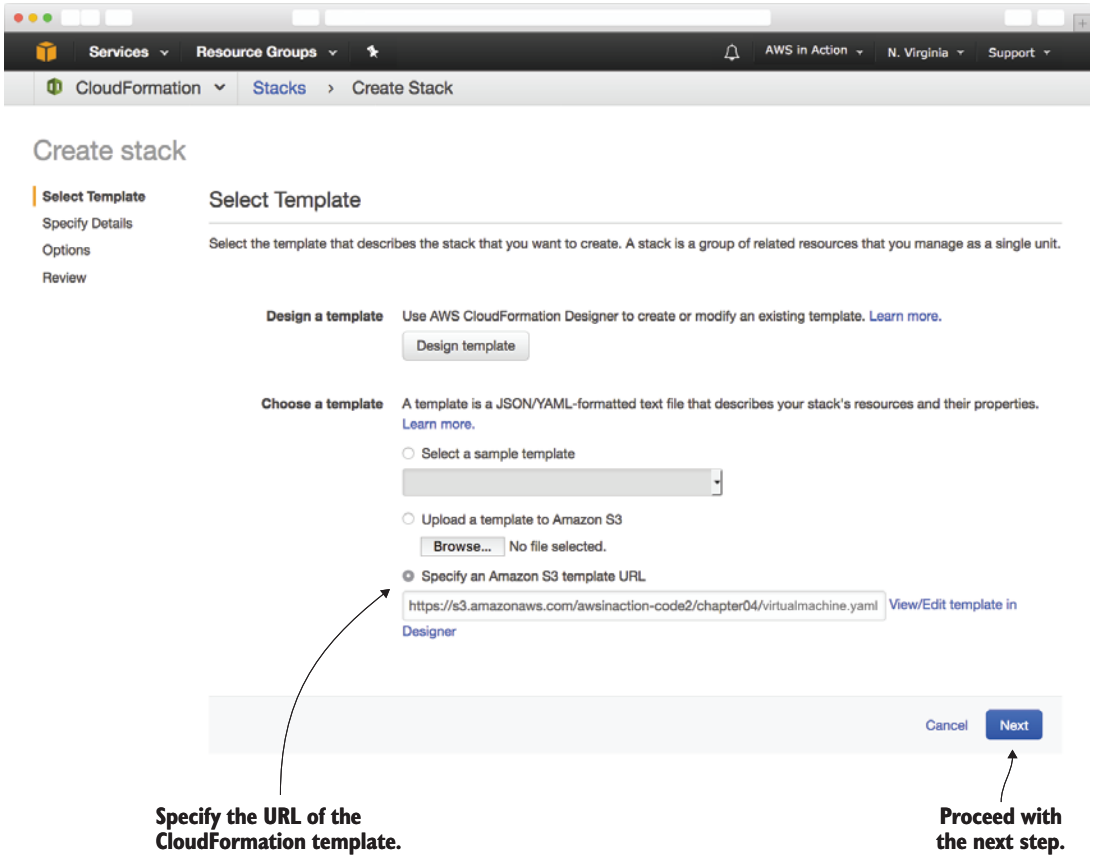


Figure 4.14 Creating a CloudFormation stack: selecting a template (step 1 of 4)

In the second step, you define the stack name and parameters. Give the stack a name like `server` and fill out the parameter values:

1. `InstanceType`—Select t2.micro.
2. `KeyName`—Select mykey.
3. `Subnet`—Select the first value in the drop-down list. You'll learn about subnets later.
4. `VPC`—Select the first value in the drop-down list. You'll learn about VPCs later.

Figure 4.15 shows the parameters step. Click Next after you've chosen a value for every parameter, to proceed with the next step.

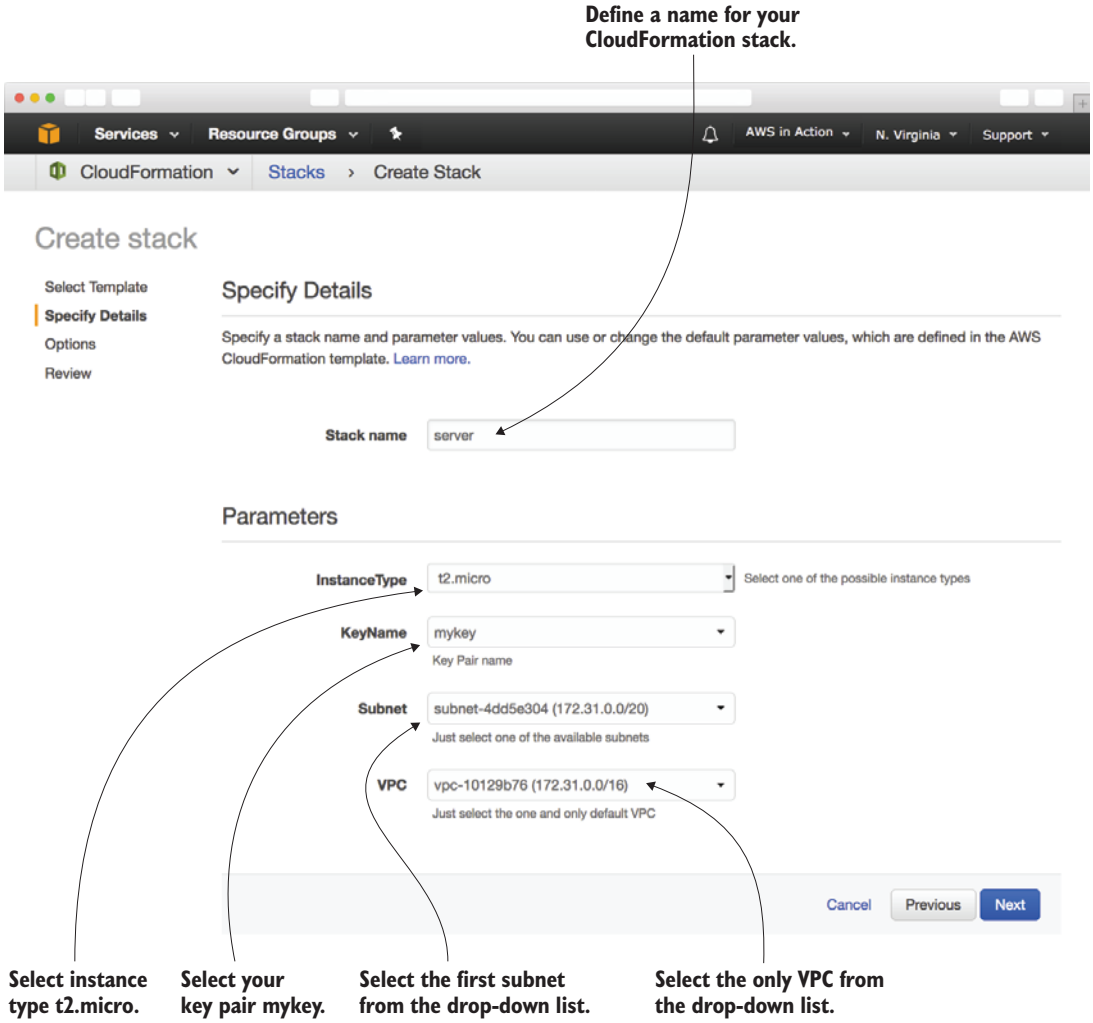


Figure 4.15 Creating a CloudFormation stack: defining parameters (step 2 of 4)

In the third step, you can define optional tags for the stack and advanced configuration. You can skip this step at this point in the book, because you will not use any advances features for now. All resources created by the stack will be tagged by CloudFormation by default. Click Next to go to the last step.

Step four displays a summary of the stack, as shown in figure 4.16.

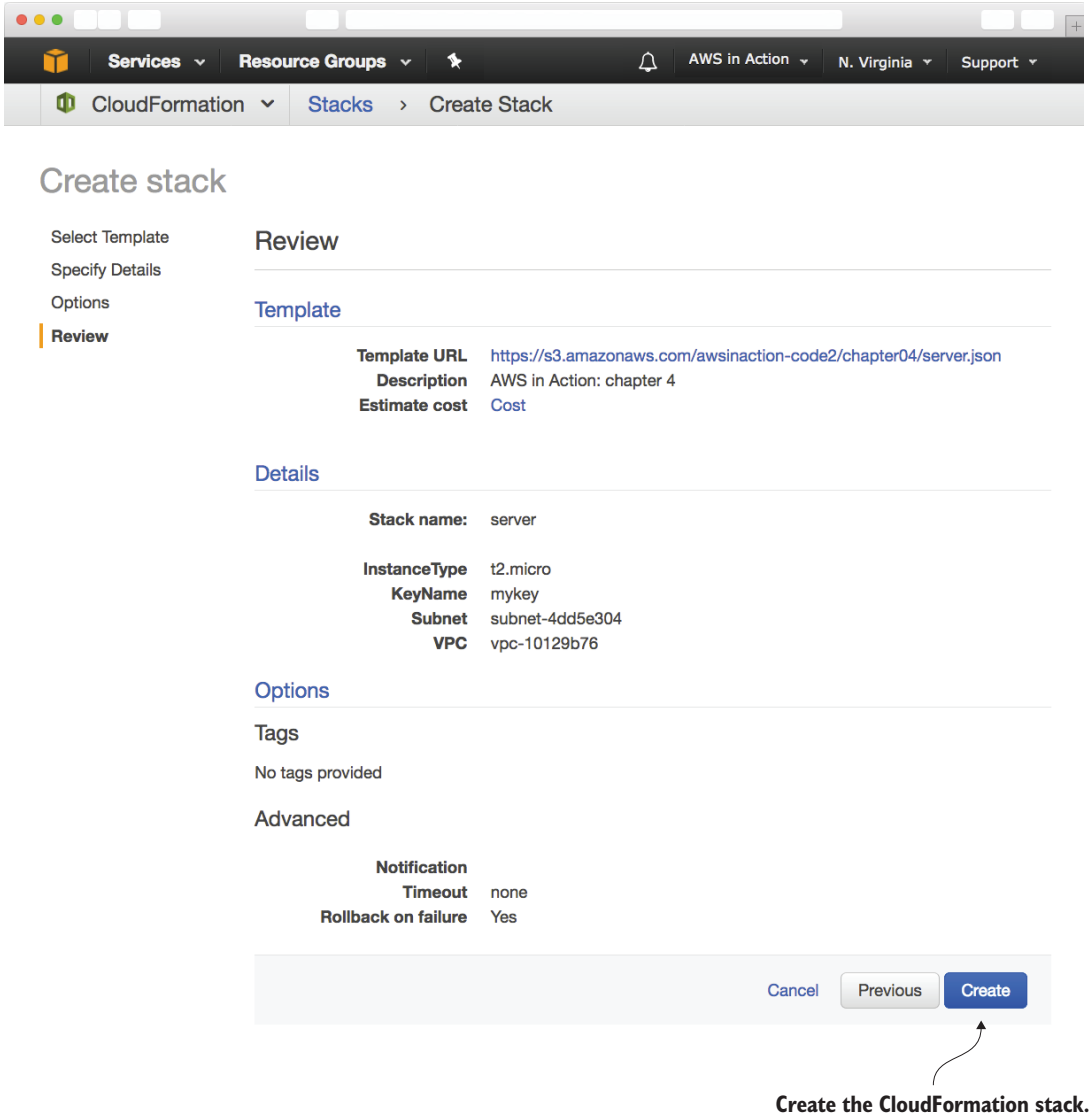


Figure 4.16 Creating a CloudFormation stack: summary (step 4 of 4)

Click Create. CloudFormation now starts to create the stack. If the process is successful, you'll see the screen shown in figure 4.17. As long as Status is `CREATE_IN_PROGRESS`, you need to be patient. When Status is `CREATE_COMPLETE`, select the stack and click the Outputs tab to see the public name of the EC2 instance.

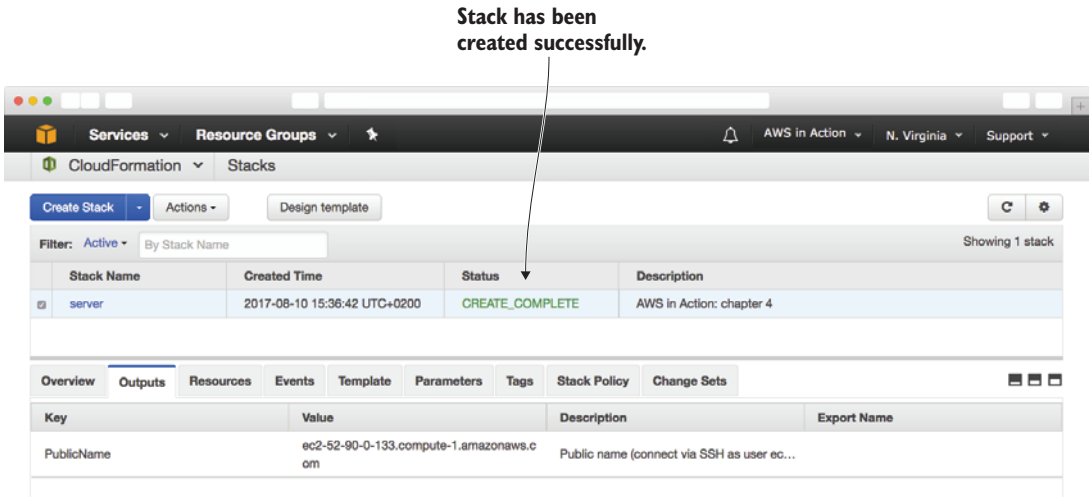


Figure 4.17 The CloudFormation stack has been created.

It's time to test modifying the instance type. Select the stack, and click the Update Stack button. The wizard that starts is similar to the one you used during stack creation. Figure 4.18 shows the first step of the wizard. Select Use Current Template and proceed with the next step.

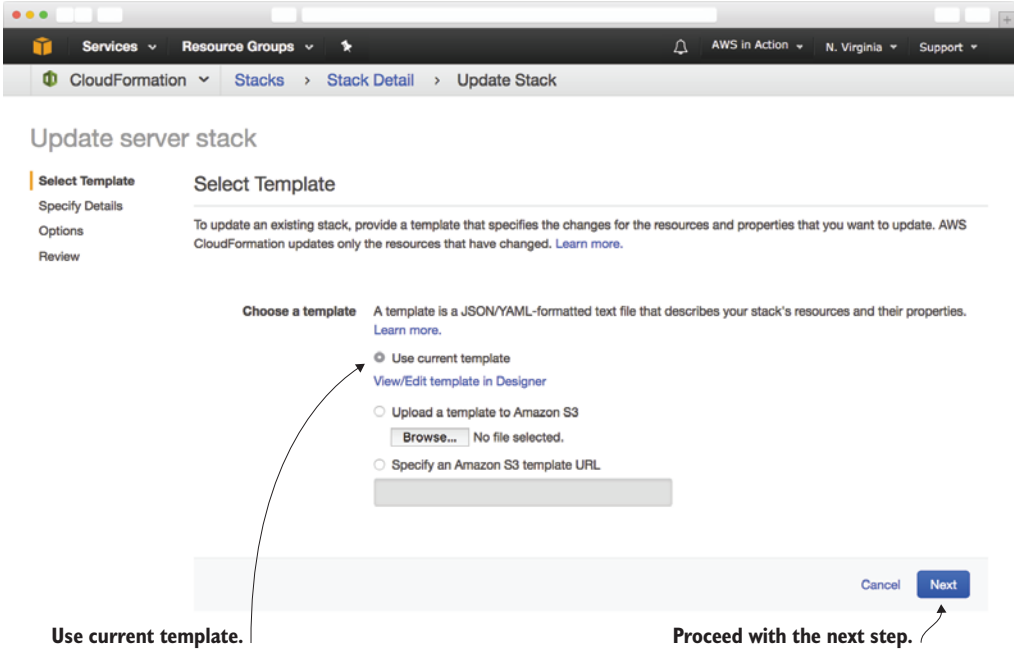


Figure 4.18 Updating the CloudFormation stack: summary (step 1 of 4)

In step 2, you need to change the InstanceType parameter value: choose t2.small to double or t2.medium to quadruple the computing power of your EC2 instance.

> **WARNING** Starting a virtual machine with instance type t2.small or t2.medium will incur charges. See http://aws.amazon.com/ec2/pricing to find out the current hourly price.

Step 3 is about sophisticated options during the update of the stack. You don't need any of these features now, so skip the step by clicking Next. Step 4 is a summary; click Update. The stack now has Status `UPDATE_IN_PROGRESS`. After a few minutes, Status should change to `UPDATE_COMPLETE`. You can select the stack and get the public name of a new EC2 instance with the increased instance type by looking at the Outputs tab.

Alternatives to CloudFormation

If you don't want to write plain JSON or YAML to create templates for your infrastructure, there are a few alternatives to CloudFormation. Tools like Troposphere, a library written in Python, help you to create CloudFormation templates without having to write JSON or YAML. They add another abstraction level on top of CloudFormation to do so.

There are also tools that allow you to use infrastructure as code without needing CloudFormation. Terraform (https://www.terraform.io/) let you describe your infrastructure as code, for example.

When you changed the parameter, CloudFormation figured out what needed to be done to achieve the end result. That's the power of a declarative approach: you say what the end result should look like, not how the end result should be achieved.

Cleaning up

Delete the stack by selecting it and clicking the Delete Stack button.

Summary

- Use the CLI, one of the SDKs, or CloudFormation to automate your infrastructure on AWS.
- Infrastructure as code describes the approach of programming the creation and modification of your infrastructure, including virtual machines, networking, storage, and more.
- You can use the CLI to automate complex processes in AWS with scripts (Bash and PowerShell).

- You can use SDKs for nine programming languages and platforms to embed AWS into your applications and create applications like nodecc.
- CloudFormation uses a declarative approach in JSON or YAML: you only define the end state of your infrastructure, and CloudFormation figures out how this state can be achieved. The major parts of a CloudFormation template are parameters, resources, and outputs.